Xede: Practical Exploit Early Detection

Meining Nie[1], Purui Su[1,2(✉)], Qi Li[3], Zhi Wang[4], Lingyun Ying[1], Jinlong Hu[5], and Dengguo Feng[1]

[1] Trusted Computing and Information Assurance Laboratory, Institute of Software, CAS, Beijing, People's Republic of China
purui@iscas.ac.cn
[2] State Key Laboratory of Computer Science, Institute of Software, CAS, Beijing, China
[3] Tsinghua University, Beijing, China
[4] Florida State University, Tallahassee, USA
[5] South China University of Technology, Guangzhou, China

Abstract. Code reuse and code injection attacks have become the popular techniques for advanced persistent threat (APT) to bypass exploit-mitigation mechanisms deployed in modern operating systems. Meanwhile, complex, benign programs such as Microsoft Office employ many advanced techniques to improve the performance. Code execution patterns generated by these techniques are surprisingly similar to exploits. This makes the practical exploit detection very challenging, especially on the Windows platform. In this paper, we propose a practical exploit early detection system called Xede to comprehensively detect code reuse and code injection attacks. Xede can effectively reduce false positives and false negatives in the exploit detection. We demonstrate the effectiveness of Xede by experimenting with exploit samples and deploying Xede on the Internet. Xede can accurately detect all types of exploits. In particular, it can capture many exploits that cannot be captured by mainstream anti-virus software and detect exploits that fail to compromise the systems due to variations in the system configurations.

Keywords: Exploits · Code injection · Code reuse · ROP · Detection

1 Introduction

Advanced persistent threat (APT) is a stealthy, continuous, and targeted attack against high-value targets, such as enterprises and government agencies. It is often motivated by major financial or political reasons. There are a stream of recent infamous attacks that cause vast consumer data breach and other disastrous consequences [4–6]. APT has since become a major security concern to these organizations. APT often employs zero-day (or recently-disclosed) vulnerabilities in popular programs, such as Microsoft Office, Internet Explorer, Adobe Flash, and Adobe Acrobat [37,40], to penetrate the defenses of its target. Traditional signature-based (black-listing) malware and intrusion detection systems

H. Bos et al. (Eds.): RAID 2015, LNCS 9404, pp. 198–221, 2015.
DOI: 10.1007/978-3-319-26362-5_10

have increasingly become ineffective against APT. Meanwhile, white-listing is not only inconvenient for end users due to compatibility issues, but also incapable of catching malicious inputs (unless there is a formal definition of all valid and secure inputs). Instead, an effective defense against APT should focus on the early detection of exploits. Exploits often violate some code or control-flow integrity. For example, code injection attacks introduce new (malicious) code into the system, while return-oriented programming (ROP [39], a typical code reuse attack) manipulates the control flow to execute its gadgets, short code snippets that each ends with a return instruction. An exploit detection system checking these integrities could detect a wide spectrum of exploits.

However, the practical exploit detection is still a challenging problem, especially for the Windows systems. Remote network exploits against common Windows applications are the most prevailing attack surface [37]. Popular Windows applications, such as Microsoft Office, often employ the following advanced techniques that are surprisingly similar to exploits. If not carefully vetted, these programs could be mistakenly classified as malicious files, leading to high false positives. *First*, many large Windows functions generate dynamic code to improve performance or extend the functionality. We analyze 7 common targets in Windows and find that all these applications generate a large quantity of dynamic codes. An exploit detection system should separate the generated code from the injected malicious code. *Second*, some applications may replace or adjust the return addresses on the stack for obscure reasons. We also saw the example code that pushes return addresses directly to the stack, instead of through the `call` instructions. These irregular behaviors disrupt security mechanisms like the shadow stack expect the `call` and `return` instructions to be matched. Exploit detection systems need to accommodate these special but common program tricks to reduce false positives. *Third*, benign windows applications may have many short code sequences that resemble gadgets and are wrongfully detected as such by existing schemes. For instance, we analyzed a large amount of samples collected from the Internet, and found that most of them contain many small gadget sequences. In particular, we observed around 5,000 false positives when simple ROP detection schemes are employed to analyze one PDF file. Furthermore, commodity operating systems have incorporated exploit mitigation techniques such as data-execution prevention (DEP [17]) and address space layout randomization (ASLR [26]). These techniques significantly raise the bar for reliable exploits. Many exploits are tied to a specific run-time environment. If the detector has a different setting other than the target system, the exploits often trigger exceptions. This can and should be leveraged for the exploit detection.

In this paper, we propose Xede, a practical exploit early detection system to protect against APT. Xede can be deployed at the gateway to scan the incoming traffic, such as emails, or deployed as a web service to scan files for exploit detection. Xede has three major detection engines: exploit exception detector, code injection detector, and code reuse detector. The first component detects failed attack attempts by monitoring exceptions. Many exploits rely on the specific system configurations. Xede uses a variant of software configurations (e.g., OS

with different patching levels) to induce the instability of exploits. Our experiments reveal that around 70 % of the malware samples are unstable, causing run-time exceptions. The second component detects (malicious) injected code by comparing the executed instructions against a list of benign instructions. This list is timely updated with the legitimate dynamically generated code to reduce false positives. Code injection attacks are often combined with code reuse attacks to bypass the DEP protection. Xede's third component focuses on the code reuse detection. It can detect both the more popular return-oriented programming (ROP) attacks and jump-oriented programming (JOP) attacks. Surprisingly, our experiments show that around 20 % of exploits contain a mix of return-based and jmp-based gadgets. Xede's code reuse detector can accommodate all the previously-mentioned eccentric program behaviors. With these three components, Xede can detect many different types of exploits, including zero-day exploits. We have built a prototype of Xede for the Windows operating systems. Our evaluation demonstrates that Xede is highly effective in detecting exploits. For example, we can detect all the malware samples we collected from the Internet. We have also deployed Xede on the Internet as a public service [42] to scan user-provided suspicious files.

2 Background

In order to exploit a vulnerability of a program, the following three steps need to be performed. Firstly, attackers need to construct memory layout of the program to host shellcode and data. Secondly, the attackers hijack the control flow of the program to injected shellcode directly or by constructing ROP gadgets. Lastly, the shellcode is executed to exploit the vulnerability. Note that shellcode could be injected into memory of target processes by either direct code injection, i.e., by code injection attacks, or constructing instruction chains through a serious of ROP gadgets, i.e., ROP attacks. Nowadays it is not easy to directly construct code injection attacks since the DEP defense mechanism employed in Windows does not allow direct code injection on writable and executable memory space. To address this issue, ROP gadgets are used to construct shellcode by leveraging indirect branch instructions, i.e., ROP gadgets, in target processes. Besides ROP gadgets that usually end with ret instructions, JOP gadgets ending with indirect jmp instructions can be used to construct shellcode as well [8]. In this paper, for simplicity, we collectively call them ROP gadgets.

To launch pure code injection attacks, the attackers can arrange memory layout to host shellcode by using heap spray or stack overflow. As shown in Fig. 1, the attackers can use the `HeapAlloc` function to allocate the shellcode at the addresses of `0x06060606`, `0x0A0A0A0A`, and `0x0C0C0C0C`, respectively. The control flow of the program can be hijacked to the shellcode by altering the function pointer or return address, and the pointer or the address will point to the location of the injected shellcode. For instance, the function pointer in Fig. 1 is changed to `0x0C0C0C0C` that is the location of the injected shellcode. Once the altered function pointer is invoked, the shellcode will be executed. Unlike

code injection attacks, ROP attacks identify ROP gadgets and construct the stack including the addresses of the ROP gadgets. As shown in Fig. 1, gadgets are located at different locations, e.g., at `0x5e861192` and `0x5e81372a`. When the ESP register points to the address of the first gadget, i.e., `0x0x5e861192`, the control flow of program will be redirected to the gadgets by leveraging a ret instruction. The gadgets are executed one by one according to the addresses stored on the stack. Eventually, the `WriteProcessMemory` function is called to finish exploit execution.

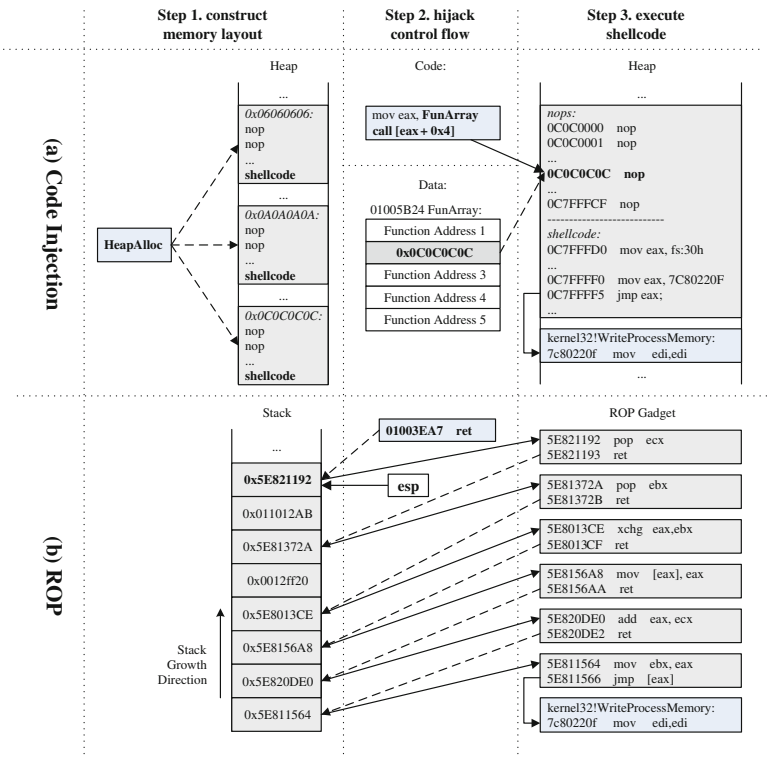

Fig. 1. Exploits examples with different exploitation techniques.

Normally, code injection attacks are easier to construct. However, the data execution prevention (DEP) mechanism raises the bar for code injection attacks. Therefore, it is not easy to directly inject executable code into memory with DEP-enabled systems. ROP attacks are immune to DEP but can be throttled by the address space layout randomization (ASLR) mechanism. To evade these prevention mechanisms, attackers adopt hybrid approaches to launch attacks, i.e., they can construct ROP gadgets to bypass the prevention mechanisms and leverage code injection attack to execute shellcode.

Key Observation. Benign programs contain some attack patterns, e.g., dynamic code, mismatching of call and return instructions, and small gadget sequence, which make exploit detection harder. However, exploits generated by different attack techniques share a common pattern that they redirect the control flow to some abnormal places other than the original ones. Specifically, the control flow is redirected to the pre-constructed shellcode or the first ROP gadget. Hence, we could detect different exploits by detecting unexpected jumps according to different attack features.

3 System Design

In this section, we describe the design of Xede, a practical exploit early detection system, in detail.

3.1 Overview

Xede is a comprehensive exploit detection platform. It can detect both code injection and code reuse attacks. Code injection attacks introduce alien code into the system. Xede accordingly builds a list of benign code and detects branches to the injected code by comparing branch destinations to that list. Meanwhile, code reuse attacks like return-oriented programming (ROP) have distinctive control flow patterns. For example, ROP reuses short snippets of the existing code called gadgets. Each gadget ends with a return instruction which "returns" to the next gadget. As such, ROP has a sequence of unbalanced returns. Xede can thus detect code reuse attacks by looking for these control flow patterns. In addition, the ubiquitous deployment of exploit mitigation mechanisms, such as DEP and ASLR, has significantly raised the bar for working exploits. Many exploits become unreliable as a result of that. This observation is leveraged by Xede to heuristically detect exploits by monitoring "abnormal" exceptions.

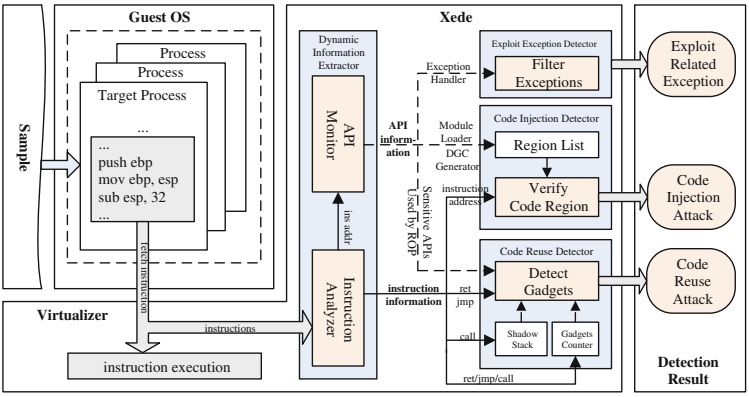

Fig. 2. The architecture of Xede

Figure 2 shows the architecture of Xede. Xede can be based on a whole-system emulator like QEMU or a dynamic binary translation based virtualization platform, such as VirtualBox and VMware workstation. At run-time, the virtualizer feeds the details of executed instruction to Xede. Xede has four major components. A dynamic information extractor extracts the run-time information of the running system, such as the executed instructions, exceptions, and the loaded modules. That information is passed to the three exploit detection engines: exploit exception detector, code injection detector, and code reuse detector. They try to detect exploits with abnormal exceptions, injected instructions, and characteristic code-reuse control flow patterns, respectively. In the rest of this section, we will describe each module.

3.2 Dynamic Information Extractor

Xede is based on a system emulator. This allows Xede to monitor every aspect of the target system. Xede is particularly interested in the details of certain executed instructions and critical API calls. The emulator passes the executed instructions and their operands to Xede. If it is a branch instruction, Xede checks whether the branch target points to a benign code block to detect injected code. Moreover, Xede uses the virtual machine introspection technology [24] to reconstruct the high-level API calls. Xede is interested in three types of API functions: the functions that load a kernel module or a shared library, the functions that handle exceptions, and the functions that are often misused by code reuse attacks (e.g., those that change the memory protection). The parameters and return values of those API calls allow Xede to identify valid code regions, catch abnormal exceptions, and detect code reuse attacks.

3.3 Exploit Exception Detector

Most commodity operating systems support two exploit mitigation mechanisms: DEP and ASLR. The former prevents code from being overwritten and data from being executed. Accordingly, no injected code can be immediately executed. It must be made executable first. The latter randomizes the layout of a process to prevent the attacker from locating useful gadgets. Many exploits have severe compatibility issues. They often trigger exceptions when the target software configurations change. Because of these issues, exploits are significantly harder to be perfect. On the other hand, popular attack targets, such as Microsoft Office, Adobe Acrobat and Flash, and Oracle Java virtual machine, all run fairly stable under normal operations. An exception in these programs may signify an ongoing attack. Therefore, Xede tries to detect *failed* exploits by monitoring the exceptions caused by these programs. Most existing exploit detection systems focus solely on detecting *successful* exploits. Xede instead can detect both successful and failed exploits.

There are two challenges to this approach: *first*, a process may cause various benign exceptions during its execution. For example, the kernel may swap out a part of the process to relieve the memory pressure. If that part is accessed

by the process, an exception will be raised by the hardware. Hence, we must be able to distinguish these benign exceptions from the ones caused by the attacks. *Second*, complex commercial programs like Microsoft Office often try to handle exceptions if they can to provide a smooth user experience. There are many different ways to handle an exception. This could confuse the exploit detection systems. Therefore, we need to have a single unified method to catch exceptions. There are 23 different exceptions in the Windows operating system roughly in the following five categories: memory-related, exception-related, debugging-related, integer-related, and floating-point-related. Exceptions caused by exploits most likely fall into the first category. For example, they may read or write invalid data areas or execute illegal instructions. Memory-related exceptions are handled by the page fault handler in the kernel (i.e. the `MmAccessFault` function in Windows), which may further deliver them to the faulting user process.

To address the first challenge, we need to separate benign exceptions from ones caused by attacks. Programs can cause benign exceptions in the following two scenarios: *first*, the kernel uses demand paging to reduce memory consumption. For example, it may load a part of the process address space lazily from the disk, or swaps some memory pages out to the disk if they have not been used for a long time. *Second*, the user process itself might use memory-related exceptions to implement lazy memory allocation. For example, some programs use large data containers with an unknown length. The memory is only allocated when the data is accessed and an exception is raised. Microsoft Power Point 2007 uses this approach to manage Object Linking and Embedding (OLE) data. Xede has to exclude both cases from the exploit detection otherwise there will be lots of false positives. The first case is rather straightforward to exclude. The page fault handler (`MmAccessFault`) recognizes that this page fault is caused by a valid-but-not-present page. It reads the accessed data from the backing store and returns `STATUS_SUCCESS` to restart the instruction. Exceptions caused by attacks instead cause `MmAccessFault` to return `STATUS_ACCESS_VIOLATION`. However, `MmAccessFault` also returns `STATUS_ACCESS_VIOLATION` for the second case. To solve this problem, we first compare the faulting instruction address against the list of legitimate code. An alert will be raised by Xede if the instruction is illegitimate. Otherwise, we record the faulting data address expecting the program to allocate new memory for it. The next time a new data region is allocated, we check whether it covers the previous faulting data address. If so, the exception is considered to be benign.

3.4 Code Injection Detector

Even though modern operating systems like Windows enforce data execution prevention, code injection is still possible. For example, some (old) programs or libraries have mixed code and data pages. These pages must be made executable and writable, violating the DEP principle. If a program can dynamically generate code, its address space could contain writable and executable pages. Moreover, the memory protection can also be changed by system calls. Xede accordingly

has a code inject detector, which builds at run-time a list of legitimate code regions and checks whether an executed instruction is in the list or not.

Legitimate Code Regions: A process consists of many different executable modules. For example, the kernel inserts the standard dynamic loader into the process to start a new one. The loader then loads the main program together with its linked shared libraries. The program itself can load additional dynamic libraries at run-time. Moreover, other processes, such as the input method editor (IME), can inject code into the process. Xede needs to identify all these executable modules. To this end, Xede hooks the API functions that may load code into a process. Their run-time parameters and return values provide the necessary information for Xede to locate the loaded executable (the program or a shared library) and know the base address of the executable. Xede then parses the executable to find the offset and size of its code section. The run-time code location is the base plus the offset. Correspondingly, we also monitor the API calls that unload an executable and remove the associated code section from the list of legitimate code regions. This list is also kept up-to-date with dynamically generated code.

Dynamically Generated Code: Dynamic code generation is a popular method to improve program performance. For example, modern browsers rely on just-in-time compiling to speedup JavaScript programs. This makes it possible to run large complex applications such as Google Maps in the browser. Xede requires a simple and generic way to recognize dynamically generated code. To that end, Xede hooks the related API calls to monitor memory allocations and memory protection changes.

To generate dynamic code, a process can allocate a block of writable-and-executable memory and then write the code into it, or it can save the code in the already-allocated writable memory and calls a system API to make the memory executable. In either case, Xede hooks the memory allocation and modification APIs. If a block of memory is made executable, we add it to the list of legitimate code region list. Likewise, if a block of memory loses its execution permission or is freed, we remove it from the code region list. Note that these two methods can only allocate execute memory in the page granularity (`4KB` for x86-32). Nevertheless, there are some unsafe programs that generate code using the executable heap. That is, the whole heap is made writable and executable. It is thus unnecessary for these programs to explicitly allocate executable memory pages. They could just use the ordinary `malloc` and `free` functions to manage executable memory. A simple solution would add the whole heap section to the executable code region. This leads to a high false negative rate for Xede because code injected in the heap is mistaken as benign code. To identify the exact regions of the generated code, we observe that well-designed programs use `NtFlushInstructionCache` to flush the instruction cache if new code is generated or the existing code is modified (self-modifying code). Xede thus hooks this function and adds the memory block specified in its parameters to the benign code region list (we merge continuous regions to reduce the list size.) On architectures with relaxed cache consistency mode, the instruction cache must

be flushed for the generated/modified code to take effect. This is not strictly necessary for the x86 architecture which provides transparent instruction cache flushing. However, we expect most commercial programs (i.e., the poplar targets of attacks) to follow the correct practice to flush the cache because Windows does support several different architectures (e.g., ARM).

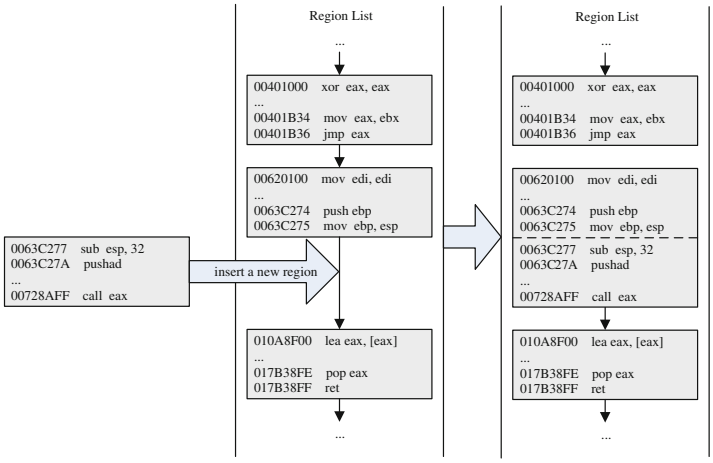

Fig. 3. Merge adjacent code regions

Code Injection Detection: Xede detects the injected code by checking whether an executed instruction lies in the list of benign code regions. However, it is prohibitively time-consuming to check this for every single instruction. Xede instead validates this property when the control flow is changed. In other words, it only checks that the destination of each branch instruction is within the code region list. This coincides with the concept of basic blocks. Each basic block is a linear sequence of instructions with only one entry point and one exit point. To guarantee correctness, we must ensure that each basic block lies within a single region. The code region list we built should not have problems in this regard if the program is correct. Figure 3 shows how this requirement is fulfilled by merging adjacent blocks of dynamically generated code. In addition, many basic blocks target another basic block in the same region. Xede thus verifies whether a branch target is within the current list, and only falls back to the whole list if that quick check fails.

3.5 Code Reuse Detector

With the wide-spread deployment of DEP and ASLR, code reuse attacks have become one of the most popular attack vectors. Fine-grained code reuse attacks include return-oriented programming (ROP) and jump-oriented programming

(JOP). ROP uses return instructions to chain gadgets, while JOP uses jump instructions instead. ROP is often used by attackers to bypass DEP. Xede can detect both ROP and JOP. Xede detects JOP by identifying sequences of gadget-like instructions. In this paper, we omit the details of the JOP detection, and focus on the more practical and more popular ROP attacks instead.

In ROP, each gadget ends with a return instruction. When a gadget returns, it pops the address of the next gadget off the stack and "returns" to it. A typical ROP attack consists of 17 to 30 gadgets [9]. This introduces a sequence of erratic return-based control flow transfers. For example, unlike legitimate return instructions that jump to a valid return site (i.e., an instruction preceded by a call instruction), gadgets often do not mimic a return site. As such, one way to detect ROP is to check whether the return target is preceded by a call instruction. Unfortunately, this method can be easily bypassed by call-preceded gadgets [7]. On the other hand, normal program execution has (mostly) balanced call and return pairs, but ROP causes mismatch between them (more returns than calls). This provides a more precise and reliable method to detect ROP. Specifically, Xede maintains a shadow stack for return addresses. It pushes the return address to the stack when a call instruction is executed, and pops the return address at the top of the stack and compares it to the actual return address when a return instruction is executed. This approach can detect ROP attacks because, when an ROP attack overwrites the stack with its gadget addresses, these addresses are not added to the shadow stack. However, it cannot be applied to the Windows platform due to various erratic behaviors of benign programs. We observe all of the following cases:

1. The program may replace the return address on stack with a completely different return address, causing the call-return mismatch.
2. The exception handling, setjmp/longjmp, and call/pop sequences introduces extra call instructions without the matching return instructions. For example, a program must be compiled as position-independent executable (PIE) to benefit from ASLR. PIE uses the PC-relative addressing mode to access its code and data. However, the x86-32 architecture does not natively support this addressing mode. Compilers instead emulate it by calling the next instruction and immediately popping the return address off the stack.
3. The program may adjust the return address on the stack (for unknown reasons), but usually within a few bytes.

As such, return addresses on the stack might be added, removed, and changed during the normal program execution. Xede needs to handle all these cases to reduce false positives.

First, to handle added return addresses, we search the shadow stack top-down for possible matches. If a match is found, we consider this return benign and pop the excessive returns above it. Note that this will not conflict with recursive functions whose return addresses might appear on the stack many times because normal recursive functions have matched call and return pairs. *Second*, to handle removed return addresses, we observe that normal program often removes only a single extra return address from the stack at a time. Therefore, Xede

only considers it an ROP attack if there are `N` consecutive mismatched return addresses. According to our observation, an exploit can be accurately captured if the enhanced shadow stack captures three consecutive mismatched return addresses. Therefore, in our prototype, we use three for `N`. A normal real-world ROP attack usually uses 17 to 30 gadgets [9], say, to arrange gadgets and store parameters. Under some rare conditions, the attacker might be able to launch an ROP attack with two gadgets, one to make the injected shellcode executable (e.g., with the `VirtualProtect` function, assuming the parameters to this function happen to be placed.) and the other to execute the shellcode. To defeat ROP attacks with a very short gadget sequence, Xede hooks 52 most common APIs used by ROP attacks and checks whether these functions are "called" by a return instruction. If so, Xede considers it an ROP attack and raises an alert. *Third*, to handle changed return addresses, we analyze a number of common executables and find that return addresses mostly change by less than or equal to 16 bytes. Therefore, if a return address does not match the return address on the top of the stack, we check whether they are within 16 bytes of each other. If so, we consider the return address has been changed by the program itself and do not raise an alert. In addition, to avoid repeating the above time-consuming heuristics, we add any detected special cases to a white list and quickly check if a potential mismatch is discovered.

Xede can also detect ROP attacks that use stack pivoting. Stack pivoting points `esp`, the top of the stack, to a buffer under the attacker's control, such as a maliciously constructed heap area. The fake stack facilitates the attacker to carry out complex ROP attacks. To detect stack pivoting, we verify whether the `esp` register points to a valid stack area when we detect a potential mismatch of return addresses. We can retrieve the base and length of the stack from the thread control blocks in the guest operating system, such as the following fields in the Windows TEB (thread environment block) structure: `teb->NtTib->StackBase` and `teb->NtTib->StackLimit`.

4 Implementation

We have implemented a prototype of Xede based on QEMU, a generic open-source emulator. QEMU allows us to flexibly instrument instructions/basic blocks and introspect the guest memory. However, the design of Xede is not tied to QEMU. It is equally applicable to other hardware emulators (e.g., Bochs) and binary-translation based virtualization systems (e.g., VMware workstation and Oracle VirtualBox).

Figure 4 shows the overall architecture of our QEMU-based prototype. QEMU parses the guest instructions and further translates them into basic blocks. Basic blocks may further be linked into super blocks (i.e., translation blocks of QEMU). As previously mentioned, Xede has four major components. The dynamic information extractor retrieves the instruction details and hooks important API calls. As such, each time a new instruction is parsed, and its information is passed to this module for bookkeeping. The module also

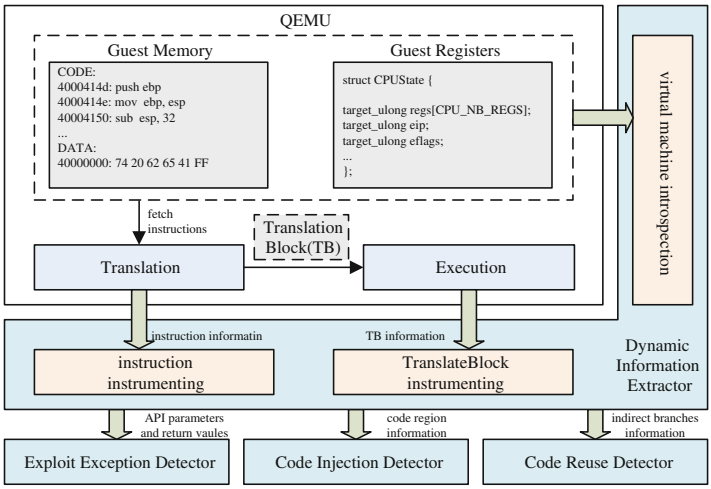

Fig. 4. Xede prototype based on QEMU

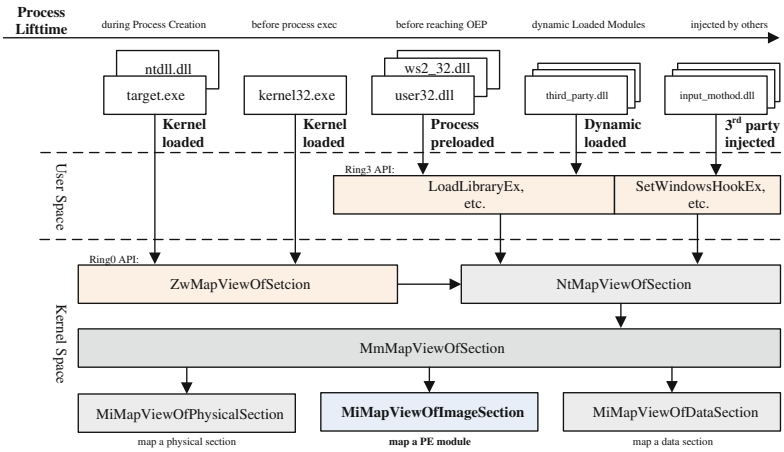

Fig. 5. Xede introspects Windows libraries

inserts a call back to the entry point of each interested API function (e.g., `NtFlushInstructionCache`) to catch its parameters and return values. The API call data is used by the second module, exploit exception detector, to detect failed exploits. To reduce the overhead of address validation, code injection detector only validates the branch targets to ensure that they jump to legitimate code regions. As such, it inserts a callback at the end of each basic block (this is where branch instructions are located.) The last module, code reuse detector, instruments indirect branch instructions (i.e., indirect calls, indirect jumps, and returns).

Figure 5 shows how our prototype for Windows intercepts the executable loading events. A process may include executables loaded by the kernel (e.g., `ntdll.dll` and `kernel32.exe`), the dynamic loader (e.g., `user_32.dll` and `ws2_32.dll`), the process itself, and libraries injected by third-party programs such as input method editors. These modules are loaded into the process using different API functions. For example, the kernel uses function `ZwMapViewOfSection` to load an executable section, and a user process can load dynamic libraries using a series of related functions such as `LoadLibraryEx`. A third-party library can be injected into the process with `SetWindowsHookEx`. However, these functions eventually converge at the `MiMapViewofImageSection` function. As such, Xede hooks this function to intercept the executable module loading events.

Xede leverages the guest kernel states to improve the preciseness of the detection. For example, it retrieves the valid stack area from the kernel to detect stack pivoting. This technology is commonly known as virtual machine introspection [20,24], which reconstructs the high-level semantics from the low-level raw data such as the memory and disk images. Our semantic analyzer is developed to perform this task.

5 Evaluation

In this section, we evaluate effectiveness of exploit detection with Xede and the incurred overheads. In particular, we systematically analyze two real exploit cases detected by Xede. We demonstrate the effectiveness of Xede by detecting real exploits collected from contagiodump [10], securityfocus [38], and exploit-db [18]. We deploy our Xede prototype as a service to detect exploits on the Internet and collect data from two systems. We integrate Xede into the mail server of a university in China which aims to detect exploits in emails, and deploy Xede on the Internet as a pubic service [42] that provides exploit detection services for Internet users. In particular, similar to VirusTotal [43], the public service is deployed as a web service so that any Internet users can scan their files by submitting the files to the website. Currently, the service allows anonymous sample submissions from the Internet for exploit detection.

5.1 Effectiveness Evaluation

Detection with Exploit Samples. We use exploits downloaded from some websites, e.g., contagiodump [10], securityfocus [38], and exploit-db [18], to evaluate the effectiveness of exploit detection. Overall we collect 12501 exploits that are included in doc/docx/rtf files, xls/xlsx files, ppt/pptx files, and pdf files. Table 1 shows the results of exploit detection. Xede accurately detects all of these exploits. Xede detects that more than 75 % exploits are generated by using the code injection techniques. In particular, among these exploits, 51.47 % exploits adopt the ROP techniques, which validates that most of exploits combine ROP and code injection techniques, and around 19.85 % exploits leverage JMP-based

Table 1. Exploit sample proportion with different exploitation techniques.

Exploit techniques	Sample proportion
code injection	75 %
ret-based gadgets	51.47 %
jmp-based gadgets	19.85 %
exploit exception	25 %

gadgets. 25 % exploits are captured because they raise abnormal execution exceptions. Furthermore, we do not observe any pure ROP attacks.

Real Deployment Detection. We collected 1,241 samples submitted by the anonymous Internet users during three months, and collected 10,144 attachments from our university email system for one month. Specifically, we selected 5,000 active users and randomly sampled their incoming emails with a rate of 3 %, and analyzed 20 popular types of the samples attached in the emails. This results in 62,500 emails and 10,144 attachments. Note that, we collected the emails before the email filters. Table 2 shows the breakdown of file types collected in real world deployment. Xede detects 136 exploits, among which 4 and 132 exploits are from the emails and the public service, respectively. Most of the exploits are pdf files and the files generated by MS office suites. They account for 30.9 % and 58.1 %, respectively. The rest are some swf files, html/htm files, and wps files. We confirm these exploits by manual analysis. Although we observe the attacks constructed by these exploits, only 44.12 % of exploits successfully succeed, which means that these exploits heavily rely on special system environments. Therefore, it is necessary to capture and detect the exploits that do not succeed to compromise the systems. Table 3 shows the success rate of different exploits. Xede can detect all these exploits no matter if they are successfully executed, which shows that the exploit detection in Xede is independent of the target system configurations. Note that, in the experiments, we do not differentiate legitimate and malicious application "crashes" because we do not observe any legitimate "crashes".

Many exploits detected by Xede cannot be captured by the existing anti-virus software. We confirm it by using some commercial virus software, i.e., Kaspersky 2015, Mcfee AntiVirus Plus, Avira Free Antivirus 2015, and Norton 2015. Overall, all these software cannot correctly detect the exploits. As shown in Table 4, Kaspersky achieves the lowest false negative that is around 15.44 %. It only detects 115 exploits out of 136 exploits, and the rest 21 exploits cannot be detected by any anti-virus software. The results reveal that many exploits can evade detection with signature matching. It demonstrates that a generic detection system is essential to detect exploits by identifying malicious operations of software.

For the 11,249 samples that Xede did not raise an alert, we used the previously-mentioned anti-virus products to cross-validate whether Xede introduced any false negatives. None of those samples were identified by them as

Table 2. Breakdown of sample file types collected in real world deployment.

Sample type	The number of email samples	The number of submitted samples	Total number
doc/docx/dot	5840	154	5994
pdf	153	241	394
swf	2	49	51
xls/xlsx	778	112	890
html/htm	80	102	182
rtf	110	120	230
ppt/pptx/pps	82	144	226
wps	58	20	78
txt/ini	2611	115	2726
jpg/png/gif	411	180	591
chm	19	4	23
Total Number	10144	1241	11385

Table 3. Success rate of different exploits.

Sample type	The number of detected exploits	Succeed exploit	Failed exploit	Success rate
doc/docx	58	43	15	74.14 %
pdf	42	4	38	9.52 %
swf	8	0	8	0 %
xls/xlsx	11	8	3	72.73 %
htm/html	6	3	3	50 %
rtf	6	1	5	16.67 %
ppt/pptx/pps	4	1	3	25 %
wps	1	0	1	0 %

Table 4. False negatives of commercial anti-virus software.

AV software	Version	Date of DB update	False negative
Kaspersky	15.0.2.361	24/05/2015	21
McAfee	18.0.204	24/05/2015	49
Avira	15.0.10.434	24/05/2015	22
Norton	22.2.0.31	24/05/2015	32

malicious. Note that false negatives are still possible if both Xede and those anti-virus products miss the attacks. Moreover, to roughly estimate how many of these 11,249 samples may be detected by existing approaches [16,30,34] as malicious (possible false positives), we recorded the suspicious patterns detected (but eventually dismissed) by Xede in these samples. Particularly, we found that 879 xls samples cause Excel to generate more than 90KB dynamic code each, and most doc samples each lead to over 4,500 mismatched call and ret instructions in Microsoft Word. All these cases may be mis-identified as malicious by existing approaches. Xede did not raise alerts for these cases.

5.2 Case Study

We analyze two different exploit samples that detected by our Xede. One sample can successfully compromise a system by leveraging the vulnerability reported by CVE-2012-0158, and the other sample leveraging the vulnerability reported by CVE-2014-1761 fails to launch the attack due to wrong system configurations.

Case 1: CVE-2012-0158. We analyze an exploit that leverages the vulnerability named with CVE-2012-0158 [14] that is a buffer overflow vulnerability in the ListView and TreeView ActiveX controls in the MSCOMCTL.OCX library. The vulnerability is leveraged against a Doc file that combines ROP and code injection technique. In order to evade Data Execution Prevention (DEP), the Doc file invokes the system call `VirtualAlloc` to allocate a block of executable memory by constructing a ROP chain, and injects the shellcode into the space. We run the exploit in Windows 7 as guest OS with Office 2003 SP1. In order to systematically analyze the exploit techniques leveraged by the exploit, we do not terminate the exploit after it is detected. Instead, we allow Xede to detect all attacks in the exploit.

ROP Detection. Xede detects 12 anomalous `return` operations. We find that the returned address by executing the first `return` instruction is `0X7c809a81` that is exactly the address of the system call `VirtualAlloc`. By analyzing the stack information, we obtain the parameters of the system call as follows. `VirtualAlloc(0x001210b0, 0x0001000, 0x00001000, 0x0000040)`. After the system call is executed, a block of executable memory is allocated. We confirm that the memory later will be injected with the shellcode. Moreover, we detect 45,039 gadget-like sequences of instructions. But, we do not find any jmp-based gadgets.

Code Injection Detection. During execution of the exploit, Xede records 53 legal code regions and 47 regions that are executable sections generated by the modules (e.g., DLL and EXE modules). As we discussed in Sect. 3, once instructions are not executed in a legal region, Xede will treat it as an attack. Overall, Xede detect 133,643 attacks. In particular, by analyzing the first five attacks, we find that the instructions are within the memory block allocated by the `VirtualAlloc` function. It means that these instructions are the code

injected by the attackers. We confirm that the code is shellcode by manual analysis. We identify several instructions that should not be invoked by Doc files, e.g., to release PE files or invoke CMD scripts.

Case 2: CVE-2014-1761. Now we analyze another Doc sample that leverages the vulnerability of CVE-2014-1761 [15] that is executed in Windows XP OS with Office 2003 SP3. When we open the sample file, the Word application crashes. We do not find anomalies by monitoring invoked APIs.

Exploit Exception Detection. Xede identifies anomaly address access at `0x909092e4`. We confirm the sample is an exploit by running in Windows 7 OS with Office 2010 SP1. The possible reason why the exploit fails is that the part of shellcode, i.e., `0x9090`, is treated as the address and the shellcode cannot be correctly located. Therefore, the exploit was not correctly executed due to the mismatched software versions.

Summary. As we observed, most exploits combine different exploitation techniques, i.e., ROP and code injection, which similar to the exploit sample above. ROP is used to evade the DEP mechanism, and the attacks finally are mounted by executing injected shellcode. According to the two exploit sample above, we show that how Xede detects exploits no matter if they can be successfully executed.

5.3 Performance Evaluation

In this experiment we evaluate the overheads incurred by Xede and the overheads during Xede bootstrapping. The experiment is performed in an Ubuntu 12.04 server with 3.07 GHz Intel Xeon X5675 CPU and 32 GB memory. We measure the overheads incurred by Xede compared with pure QEMU. As shown in Fig. 6, Xede consumes around 60 % of CPU cycles that consumed by QEMU during the bootstrapping within the 60 seconds. The possible reason is that exploit execution incurs many virtual machine introspections during Xede bootstrapping. Note that, Xede can effectively detect most exploits during this period. After the bootstrapping, Xede does not introduce extra significant CPU consumption.

We compare the CPU utilization rate and memory overhead by measuring the resource assumption in a Guest OS with 256 MB assigned memory (see Figs. 7 and 8). We can observe that Xede does not incur many CPU cycles after bootstrapping. The CPU utilization rate in QEMU with Xede and QEMU without Xede are around 0.12 % and 0.08 %, respectively. Similarly, Xede does not introduce significant memory overheads. Therefore, Xede is very lightweight. Furthermore, we measure the overheads with parallel exploit detection. Figure 9 illustrates the resource consumption with 80 Xede instances. Memory consumption is stable and the consumption rate is within 90 %. CPU utilization rate is around 6 % except some utilization rate bursts. Thus, Xede is scalable to parallel exploit detection.

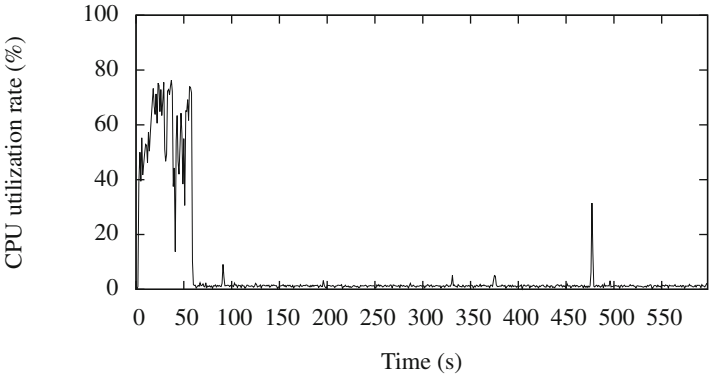

Fig. 6. Increased CPU consumption by Xede compared with QEMU.

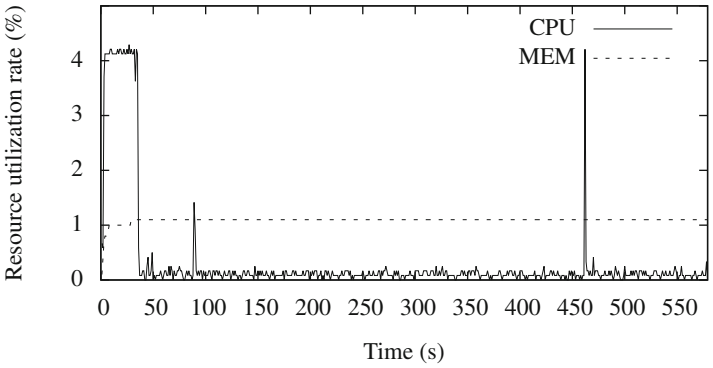

Fig. 7. CPU cycles and memory consumed by Xede.

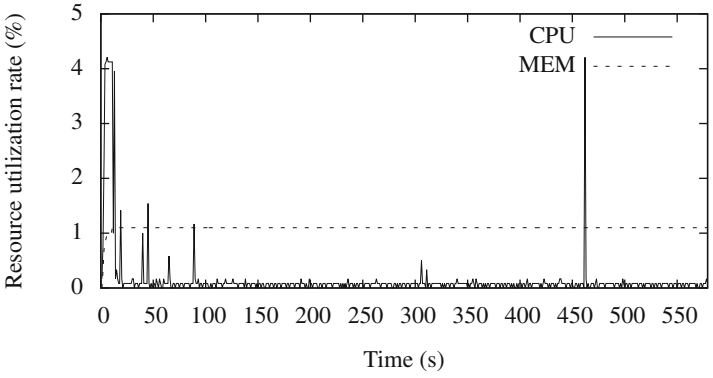

Fig. 8. Increased CPU and memory consumption by Xede compared with QEMU.

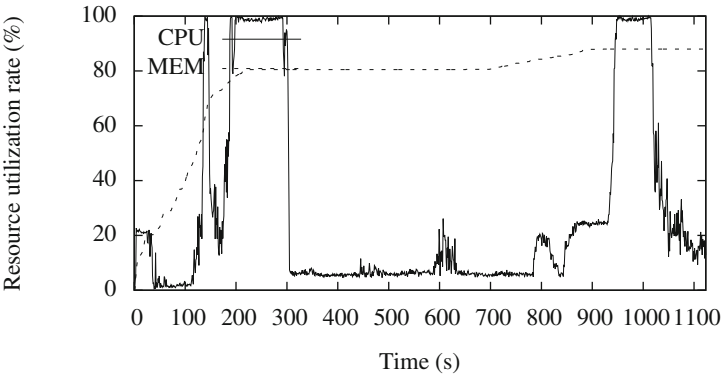

Fig. 9. CPU and memory consumption with 80 Xede instances.

6 Discussion

Detecting Exploits without ROP and Code Injection. As we observed, Xede can detect exploits leveraging ROP or/and code injection. However, it may not be able to detect exploits that hijack control flows without using ROP. For example, exploits can use the software code to copy shellcode to the legal code region and leverage legal code to hijack the control flow to the shellcode. It is very hard to construct such exploits since exploit construction requires strict conditions, e.g., writable code segment and evading DEP. For instance, we can compute checksum for different memory region to detect memory rewriting. In real practice, we do not notice any exploits that really implement this. We can easily extend Xede to detect such exploits.

Accuracy of ROP Detection. Xede does not count the number of gadgets where source and destination addresses of jmp are same. It significantly reduces the miscounted gadgets, and thereby reduces false positives of detecting JOP. However, it is possible to evade Xede by constructing gadget chains with the same intermediate gadgets, i.e., gadget *1*->gadget *x*->gadget *2*->gadget *x*->$\cdots$. In order to perform control flow hijacking from the intermediate gadgets to different gadgets, a large amount of gadgets are required to build chains between gadgets, e.g., between gadget *1* and gadget *x* in the example above. However, it is really difficult to such gadget chains, and we did not observe any attacks in real practice. Thus, we do not consider the attacks in this paper.

7 Related Work

Malicious code detection is mainly based on behavior analysis [2,3,19,27,45,46]. The principle of this technique is to monitor APIs called by the target process and then check if the process behaves properly via analyzing the API sequence. The behavior analysis techniques are also widely adopted in current

anti-virus software, such as FireEye [19] and WildFire [45]. The shortcomings of the approaches are also obvious. They need to configure corresponding behavior policies for different types of samples. By analyzing API sequence, it is very difficult either to describe the comprehensive behaviors of a benign software or to accurately define the possible behaviors of malicious code [25]. Moreover, exploits are very sensitive to the system environment. If a victim software version does not match the expected environment, the exploit will abort and the malicious behaviors cannot be identified.

Recently more researches are conducted to detect exploits by identifying shellcode [31,35,44]. Shellcode detection approaches intend to scan the content of the sample file before the file execution and then to detect whether the file includes shellcode characteristics. Polychronakis *et al.* [31] use a set of runtime heuristics to identify the presence of shellcode in arbitrary data streams. The heuristics identify machine-level operations that are inescapably performed by different shellcode types, such as reading from FS:[0x30], writing to FS:[0]. Wang *et al.* [44] blindly disassembles each network request to generate a control flow graph, and then uses novel static taint and initialization analysis algorithms to determine if self-modifying (including polymorphism) and/or indirect jump code obfuscation behavior is collected. Such line of approaches shares an important shortcoming. Content in data files is the same to the actual layout in process memory. For example, shellcode hiding in a Doc file will be parsed and reorganized by its host process, i.e., winword.exe. Therefore, it is not easy to accurately identify the presence of shellcode in different data files. Moreover, since shellcode representation may not be fundamentally different in structure from benign payloads [28], these approaches inevitably suffers from significant false positive rates.

Exploit detection by enforcing Control Flow Integrity (CFI) generates a complete control flow graph (CFG) of samples (or, the host process of the sample file if the sample is a data file) by performing static pre-analysis [1,21,47]. It monitors the execution of the target process, analyzes each instruction executed, and verifies the legitimacy of each control flow transfer by checking whether the flow transfer exists in the CFG. Zhang *et al.* [47] classify the destination addresses of indirect control flow transfer into several categories, such as code pointer constants, computed code addresses, exception handling addresses, and verify these destination addresses according to the results of static analysis. Unfortunately, the CFI approaches cannot be adopted in real systems because of the following reasons. Firstly, aiming to construct complete CFGs, CFI usually requires source code of program or debug information of whole program. The information of proprietary software is not always available. We could build CFG without those information with some tools, such as IDA [22], but the accuracy and coverage of CFG cannot be guaranteed. Secondly, the CFI approaches usually cannot verify the legitimacy of control flow transfer in dynamic code, which widely exists in modern software. Lastly, they suffers from the problems of inefficiency and high complexity [21].

Taint analysis employs a dynamic tracing technique to detect exploits [11–13, 29,33,41]. It marks input data from tainted sample, and then monitors program execution to track how the tainted attribute propagates and to check if the tainted data is used in dangerous ways. However, as far as we know, all existing taint analysis engines are unable to fully support analysis of the entire Intel instruction set. Hence, the accuracy of the analysis results cannot be guaranteed. Moreover, taint analysis needs to parse each instruction executed by the target processes, and record all addresses of tainted data, The computation complexity and complexity is not acceptable in real practice [36].

The prevention mechanisms, such as ASLR [26] and DEP [17], are adopted to protect against malicious code exploits. More exploits leverage the ROP technique to evade the mechanisms. ROP is hard to detect because it uses the existing legal instruction sequences to construct shellcode, instead of injecting shellcode. Last Branch Recording (LBR) [23], a recent technique released with Intel processors, is used to analyze the executed indirect branch instructions to see if there exists an excessively long chain of gadget-like instruction sequences. LBR-based approaches [9,30] rely on hardware for instruction-level monitoring, which introduces small runtime overhead and transparent operations. Unfortunately, these approaches have some inherent drawbacks. The LBR stack can include only 16 records, and is shared by all running processes and threads. Hence, the stack may not have enough space to record sufficient data. Moreover, these approaches cannot observe the actual path of instruction execution between two indirect jumps, thereby they cannot accurately count the number of instructions. Therefore, the LBR-based approaches may not be accurate to analyze and detect exploits. Similar to Xede, shadow stack and speculative code execution are adopted to detect ROP. For example, Davi *et al.* [16] utilized shadow stack to detect ROP. However, the system is built upon the PIN subsystem and cannot instrument the kernel code. Polychronakis *et al.* [32] used speculative code execution to analyze non-randomized modules and is unable to detect exploits leveraging randomized modules.

8 Summary

In this paper, we present the design and implementation of Xede, an exploit detection system. Xede comprehensively detect different types of exploits, e.g., generated by pure code injections, pure ROP, and hybrid exploitation techniques. We have implemented a prototype of Xede with QEMU. The evaluation demonstrates that Xede can effectively detect different exploits according experiments with samples and real world deployment on the Internet. In particular, with real world deployment, Xede detects a large number of exploits that cannot be captured by mainstream anti-virus software and exploits that raise abnormal execution exceptions due to mismatched execution environments.

Acknowledgement. We would like to thank our shepherd Christopher Kruegel, and the anonymous reviewers for their insightful comments. This work is partially supported by the National Basic Research Program of China (973 Program) (Grant

No.2012CB315804), and the National Natural Science Foundation of China (Grant No.91418206).

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, pp. 340–353. ACM (2005)
2. Amnpardaz. http://jevereg.amnpardaz.com/
3. Anubis. https://anubis.iseclab.org/
4. Flame Malware. http://en.wikipedia.org/wiki/Flame_malware
5. Sony Pictures Entertainment hack. http://en.wikipedia.org/wiki/Sony_Pictures_Entertainment_hack
6. Stuxnet. http://en.wikipedia.org/wiki/Stuxnet
7. Carlini, N., Wagner, D.: Rop is still dangerous: breaking modern defenses. In: USENIX Security Symposium (2014)
8. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 559–572. ACM (2010)
9. Cheng, Y., Zhou, Z., Yu, M., Ding, X., Deng, R.H.: Ropecker: a generic and practical approach for defending against rop attacks. In: Symposium on Network and Distributed System Security (NDSS) (2014)
10. contagiodump. http://contagiodump.blogspot.com/
11. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: end-to-end containment of internet worms. ACM SIGOPS Oper. Syst. Rev. **39**, 133–147 (2005). ACM
12. Crandall, J.R., Chong, F.: Minos: architectural support for software security through control data integrity. In: International Symposium on Microarchitecture (2004)
13. Crandall, J.R., Su, Z., Wu, S.F., Chong, F.T.: On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, pp. 235–248. ACM (2005)
14. CVE-2012-0158. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0158
15. CVE-2014-1761. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1761
16. Davi, L., Sadeghi, A.R., Winandy, M.: Ropdefender: a detection tool to defend against return-oriented programming attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 40–51. ACM (2011)
17. Data Execution Prevention. http://en.wikipedia.org/wiki/Data_Execution_Prevention
18. exploit-db. http://www.exploit-db.com/
19. FireEye. http://www.fireeye.com/
20. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the 10th Network and Distributed System Security Symposium, Febuary 2003
21. Goktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: overcoming control-flow integrity. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 575–589. IEEE (2014)

22. IDA Pro. https://www.hex-rays.com/products/ida/
23. Intel: Intel 64 and IA-32 Architectures Software Developerś Manual, Febuary 2014
24. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through VMM-based "Out-Of-the-Box" semantic view reconstruction. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, October 2007
25. Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: Accessminer: using system-centric models for malware protection. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 399–412. ACM (2010)
26. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: automated software diversity. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP 2014 (2014)
27. LastLine. https://www.lastline.com/
28. Mason, J., Small, S., Monrose, F., MacManus, G.: English shellcode. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 524–533. ACM (2009)
29. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software (2005)
30. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent rop exploit mitigation using indirect branch tracing. In: USENIX Security, pp. 447–462 (2013)
31. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Comprehensive shellcode detection using runtime heuristics. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 287–296. ACM (2010)
32. Polychronakis, M., Keromytis, A.D.: Rop payload detection using speculative code execution. In: 2011 6th International Conference on Malicious and Unwanted Software (MALWARE), pp. 58–65. IEEE (2011)
33. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. ACM SIGOPS Oper. Syst. Rev. **40**, 15–27 (2006). ACM
34. Rabek, J.C., Khazan, R.I., Lewandowski, S.M., Cunningham, R.K.: Detection of injected, dynamically generated, and obfuscated malicious code. In: Proceedings of the 2003 ACM Workshop on Rapid malcode, pp. 76–82. ACM (2003)
35. Ratanaworabhan, P., Livshits, V.B., Zorn, B.G.: Nozzle: A defense against heap-spraying code injection attacks. In: USENIX Security Symposium, pp. 169–186 (2009)
36. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 317–331. IEEE (2010)
37. Secunia: Secunia vulnerability review 2015. Technical report, Secunia (2014). http://secunia.com/vulnerability-review/
38. securityfocus. http://www.securityfocus.com/
39. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, October 2007
40. Snow, K.Z., Monrose, F.: Automatic hooking for forensic analysis of document-based code injection attacks (2012)
41. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. ACM Sigplan Not. **39**, 85–96 (2004). ACM
42. TCA Malware Analysis platform. http://www.tcasoft.com/

43. VirusTotal. https://www.virustotal.com/
44. Wang, X., Jhi, Y.C., Zhu, S., Liu, P.: Still: exploit code detection via static taint and initialization analyses. In: 2008 Annual Computer Security Applications Conference, ACSAC 2008, pp. 289–298. IEEE (2008)
45. WildFire. https://www.paloaltonetworks.com/products/technologies/wildfire.html
46. XecScan. http://scan.xecure-lab.com/
47. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: Usenix Security, pp. 337–352 (2013)